Temporal Debugging: Automating Time Travel Debugging with URDB

Ana-Maria Visan
Northeastern University
amvisan@ccs.neu.edu

Artem Y. Polyakov
Siberian State University of Telecommunications and Informatics
artpol84@gmail.com

Xin Dong
Northeastern University
xindong@ccs.neu.edu

Kapil Arya Tyler Denniston Praveen S. Solanki Gene Cooperman
Northeastern University
{kapil,tyler,psolanki,gene}@ccs.neu.edu

Abstract

This work addresses two classical problems in debugging. First, while some excellent reversible debuggers have been built for C, C++, Java, Standard ML, other languages including MATLAB, Python and Perl lack such reversible debuggers. To solve this, this work contributes a new temporal debugging approach and a new software package, URDB (Universal Reversible DeBugger), which can extend the native debuggers of these three languages into reversible debuggers. URDB also supports reversibility of the traditional gdb debugger for C and C++. Reversibility can be added to the native debugger of a new language in less than a day. URDB uses a checkpoint-restart procedure based on DMTCP (Distributed MultiThreaded CheckPointing).

The second classical debugging problem concerns long-running computations. Such computations are often difficult to debug because the initial fault may occur much earlier than the resulting error. In a long-running computation, it is important that the application execute at full native speed, with little or no slowdown due to logging of events, interpretation of code within a debugger, etc. Moreover, in the case of logging, there is an issue both of time to log many events, and also of storage to hold many events over a long process lifetime. The solution, and another contribution of this work, is that of temporal debugging. A binary search is performed over the process lifetime to find the event or point in time representing the initial fault that eventually generated the visible error.

The abilities of URDB and DMTCP/ptrace are demonstrated on a real-world C++ program designed using the Booch (Unified) methodology and currently extending to 750,000 lines: Geant4. Since long-running computations like Geant4 often use C or C++ for efficiency, it was also necessary to extend a debugger for these languages. Gdb was the natural debugger of choice for C and C++. However, since gdb is based on ptrace, it was necessary to extend DMTCP into DMTCP/ptrace. DMTCP/ptrace is the first package that can directly checkpoint a gdb session, including both the gdb process and the target application process. As an additional demonstration temporal debugging is used to debug an X-windows graphics program, xcalc, written in the C language.

1. Introduction

Programmers have long struggled under the curse of temporality in debugging complex code. An unknown program fault corrupts the logic of the program, but the resulting bug may be exposed only much later in that program. Debuggers assist one in analyzing the logic near the error, but only in order to build a hypothesis that might explain the original cause or fault of the bug. Upon producing a hypothesis, the programmer again executes the program under debugger control with the hope of obtaining more conclusive proof that the hypothesis is correct.

This style of *iterative debugging* is only a little better than print-style debugging. A better solution would be a style of *temporal debugging*. The debugger would employ *temporal search* in order to search both backwards and forwards over the process lifetime in order to verify or falsify the hypothesis about the fault. The entire process lifetime is made available for analysis by the debugger.

Such a valuable *temporal search* technique is outside the current state of the art.

1.1 Statement of Problem

One reason for the lack of temporal search is that the program fault and error are often widely separated in time. While a number of reversible debuggers have been developed over the last 40 years (see Section 2), all of these reversible debuggers have been based on some form either of logging of instructions or of events for later replay, or of both. Such logging requires large amounts of storage, and so reversible debugging has until now usually been limited to a span of time short enough so that the debugger log does not overflow memory. IGOR [FB89] is not limited by large amounts of logging, but it lack temporal search for other reasons.

In order to make this vision work, several challenges had to be met.

1. *Universality:* works with many debuggers, and can be adapted to a new debugger in less than a day

2. *Temporal search:* automatically execute binary search on a process lifetime to find the fault that was responsible for a later error (exposed bug)
3. *Practicality:* works on bugs found in real-world applications
4. *Native execution speed:* no slowdown due to logging or interpreter-based debuggers
5. *Transparent reversibility:* addition of reversibility without modification to the original debugger, the language (whether compiled or interpreted), or the operating system.
6. *Support for ptrace-based debuggers:* debuggers for compiled languages, such as gdb, usually require support for ptrace

1.2 Contribution

This work presents a new approach to temporal debugging. To validate our approach, we implemented a new temporal debugger, URDB (*Universal Reversible Debugger*), that enables such technologies as temporal search. Using URDB, we demonstrate a practical method using temporal debugging.

This work represents four points of novelty:

1. a methodology to add reversibility to any existing text-based debugger with a few hours work;
2. a temporal debugging approach for debugging long-running programs with fault and error displaced far from each other in time; and
3. the support for checkpointing entire ptrace-based debugging sessions (both debugger and target process), which was a prerequisite for the support for gdb and C/C++.
4. a practical form of bisimulation used to resolve bugs introduced when a new version of complex software is introduced.

URDB is freely available as open source software [URD09]. It currently has four *personality* modules, which support:

gdb, MATLAB, python (pdb), and perl (perl -d).

A personality for a new debugger can be developed in a few hours by modifying a 250-line python template. As a demonstration of this, URDB was used to add reversibility to the classic curses-based BSD UNIX games of rogue and robots.

The support for MATLAB is especially noteworthy, since MATLAB is closed source. We did not have access either to the source code for MATLAB itself, or for the MATLAB debugger. URDB is implemented as a "wrapper" around the target debugger. The wrapper consists of a relatively thin *debug monitor* to implement an algorithm for reversible execution (see Section 3.2). No source code for the target debugger is required, and the target debugger is not modified.

It is practical to use URDB to debug computations that may continue for days or even weeks on CPU-intensive programs. This is because the need to log user I/O and other non-deterministic system calls is usually quite moderate.

A further novelty is the demonstration of a practical form of *bisimulation* for use on real-world programs. This extends the concept of a reversible debugger, to allow one to move forward and backwards through two versions of the same program in tandem. There are large software development efforts in which complex software goes through multiple revisions. Regressions and other bugs can appear in a new version of the program. Using a modified version of URDB for bisimulation, one can move temporally through two programs simultaneously, while examining, for example, their stacks. When the higher call frames on the stack (those that are unrelated to the modifications implemented for the new version of the program) no longer match, then one has found the bug created in the new version.

Also, URDB is deterministic. In order to achieve determinism at re-execution, we log the results of system calls and I/O. By default, logging of I/O and the results of system calls is turned off. For file I/O, we provide two options. One can either: (i) log the file I/O; or (ii) make a copy at checkpoint time of the files used by the program. If all output to files is sequential, a third option supported by DMTCP/ptrace is that at restart-time, one can truncate a copy of any open files to the offset that had existed at checkpoint-time.

The temporal search abilities of URDB are demonstrated by the implementation of *reverse expression watchpoints*. A bug is isolated, and the user has determined an expression that changes from "correct" to "incorrect" at the point where the bug is exposed. One wishes to find a statement and corresponding event at which the expression was correct, but became incorrect upon evaluation of the statement. By executing a binary search on the process lifetime, one can move to such an event, and examine the bug within the debugger at a point in time where the bug is created.

For example, the user is debugging a bounded linked list. If the user program aborts with the report `linked_list_len(X) > N`, then in a normal debugging process, the user would add frequent assert statements to detect when that expression becomes true. But frequent calls to `linked_list_len()` is computationally expensive. So in URDB, the user executes a reverse expression watchpoint on the expression `linked_list_len(X) < N` to discover the last instruction for which the consistency condition was still true. This type of approach can be especially valuable for such notoriously error-prone tasks as writing a garbage collector.

Finally, URDB gains its temporal feature through the use of DMTCP/ptrace, a new version of DMTCP with support for the ptrace system call. The ptrace extension is necessary to support gdb and other symbolic debuggers for compiled binary code that use the ptrace system call as their basis.

DMTCP/ptrace is itself one of the points of novelty of this work. It is the first checkpointing package with the capability of checkpointing an entire gdb debugging sesssion: both the gdb process and the target application process. Among the novelties of DMTCP/ptrace are: a wrapper around the "ptrace" system call and tid virtualization, as described in Section 5.

1.3 Outline of Paper

Section 2 presents a brief history of reversible debugging. Section 3 presents the architecture of URDB. Section 3.1 provides background information on DMTCP. Section 3.2 describes the reversibility algorithms of the URDB monitor. Section 3.3 describes the URDB implementation. Temporal search, and in particular, reverse expression watchpoints, are described in Section 4. Section 5 describes the implementation required to extend DMTCP to DMTCP/ptrace. Finally, Section 6 presents the experimental results. That section also contains a description of bisimulation and its application to Geant4. The conclusion and future work is in Section 7.

2. Current Approaches

Reversible debuggers (sometimes called *time-traveling debuggers*) have existed at least since the work of Grishman in 1970 [Gri70] and Zelkowitz in 1973 [Zel73]. These debuggers are based on logging and reverse execution of individual instructions: *record/reverse-execute*. This first approach was further developed through such landmarks as the reversible debugger for Standard ML by Tolmach and Appel [TA90, TA95], and the recent reversible execution feature of gdb-7.0.

About five years ago, a second approach, *record/replay* was implemented with virtual machines using record/replay technology. This work was based on virtual machine snapshots and event logging. It was first demonstrated in the work of King et al. [KDC05],

followed by VMware's Lewis et al. [LDC08], and still other examples.

Both record/reverse-execute and record/replay emphasize the advantages of deterministic replay. Both approaches have difficulties in supporting SMP multi-core architectures, where logging a serialization of synchronous instructions appears to be difficult. The approach of this paper, checkpoint/re-execute is described third. Finally, a fourth approach, post-mortem debugging was made practical about three years ago, and is also discussed.

2.1 Record/reverse-execute:

Instruction logging records the state as each instruction is executed. The logging contains sufficient information to "undo" an instruction. In addition to logging instructions, one can log external I/O, signals, and other events, for better determinism. While the benefits are clear, there are also significant disadvantages. The need for logging prevents a debugger from executing code at full native speed. Further, the size of the log files can also be significant. Finally, sequential logging of multiple threads is difficult without operating system support.

An early example of such a reversible debugger was the AIDS debugger built by Grishman [Gri70] in 1970. It was used to debug FORTRAN and assembly language on the Control Data 6600 mainframe supercomputer. A similar example was the work of Zelkowitz [Zel73] in 1973. A notable continuing success using this approach is the work of Appel and Tolmach [TA90, TA95], on a reversible debugger for Standard ML.

Gdb-7.0 includes a reversible debugging capability based on instruction logging [GDB]. It presents an excellent interactive user feel, whether single-stepping or reverse-stepping through a C program. Nevertheless, one presumes that gdb-7.0 reversible debugging was designed for relatively short runs that would normally execute over seconds or minutes.

In order to more quantitatively measure the pros and cons of instruction logging, we tested gdb-7.0. Our experiments show that gdb-7.0, when running in the forward direction with logging (`target record`), gdb was measured on a simple C program for hashing at 96,873 times slower than the program running without any debugger. The average memory consumption per C statement for logging was measured at 104 bytes per program statement.

2.2 Record/replay:

This approach is traditionally implemented through virtual machines. Snapshots record the state of the machines at given intervals. A reproducible clock is achieved through values of certain CPU registers, such as the number of loads and stores since startup. This allows asynchronous events to be replayed according to the time of the original clock when they occurred. Since 2005, at least two virtual machine-based reversible debuggers have been developed: for gdb [KDC05] and for Visual Studio with C/C++ [LDC08]. Snapshots have the huge advantage that forward execution can be extremely fast, running much closer to full native speed, if there are few events to log. This is because this approach need not log each instruction, but only external events.

The need for event logging, with its associated overhead may never be removed from this approach. The execution depends on precisely recording things like the timer interrupt, along with sensitive timing at the instruction level. If external events are replayed differently from how they were originally recorded, the interval timer used by operating systems for context switching is thrown off, and processes are context switched at the wrong time. This results in a cascade of errors as processes interact with each other at the wrong times.

Additionally, the cost of virtual machine snapshots is high. They typically require about 100 MB of disk space and 30 seconds, as opposed to a typical 10 MB and less than 1 second for DMTCP to checkpoint a gdb session. (The cost of DMTCP checkpoints is expected to become still cheaper with the planned introduction of incremental checkpoints.)

2.3 Checkpoint/re-execute:

The natural level of interaction for this approach is at the process level: intermediate between the instruction level of record/reverse-execute and the operating system level of record/replay. URDB employs checkpoint files on disk. However, an alternative strategy would be *live checkpoints*. In this approach, checkpoints are created by forking of a child process of the debugged application. Process-based checkpointing is limited by the number of processes that one can simultaneously maintain.

An early, landmark approach to reversible debugging was the IGOR system of Feldman and Brown [FB89]. That implementation relied on modifying the compiler, library and loader. They also implement a custom object code interpreter (essentially a symbolic debugger) that *simulates* execution forward from a checkpoint. The object interpreter can use a selection criterion, such as `x>0`, and "simulate" (interpret) the object code in the forward direction until the desired selection criterion is met. This last capability is different from our reverse-watch command. It requires a person to choose an appropriate checkpoint event before the selection criterion occurs (before `x>0`), but a checkpoint event that is also still close enough to the time of the selection criterion that it will find the selection criterion in reasonable time. It does not use a binary search algorithm, as described here in Section 4. Further, it is limited to the C language and IGOR's object interpreter (symbolic debugger). In the approach here, URDB (using DMTCP/ptrace) is not limited to any particular language or debugger.

Gdb-6.8 and Srinivasan et al. [SKAZ04] have implemented *live checkpoints*. These can be used to execute a limited form of a reverse-continue command (restore previous checkpoint, and forward continue until next to last breakpoint). Further the ocamldebugger [LDG+08, Part III, Chapter 16] for Objective Caml combines live checkpointing with the method of Appel and Tolmach [TA95] for reversible debugging of Standard ML.

2.4 Post-mortem debugging:

A notable recent approach to reversible debugging is that of an omniscient debugger or post-mortem debugger. In this approach, a database on disk is created that logs all events of interest. Debugging can then be done after the process of interest has terminated. Only the database of events is required. The Omniscient Debugger of Pothier et al. [PT09, PTP07] appears to be the first implementation to make this approach practical.

That work requires an average of 190 bytes per event, with 470,000 events being generated per second [PTP07]. Note that this represents storing 89 megabytes per second (the product of the two preceding quantities). This is approximately the bandwidth of disk. Hence, this approach appears to be limited by the speed of disk. This problem could be alleviated by the use of parallel RAID disks, but that is an expensive solution. Furthermore, 470,000 events is likely to be significantly slower than the native execution speed of the JVM.

The Omniscient Debugger uses an instrumented Java Virtual Machine (JVM) as part of TOD (Trace-Oriented Debugger for Java). It also supports the use of partial traces in order to allow initial execution to proceed faster while dumping to the database only the events of interest.

In contrast, URDB operates on many languages, including C and C++. URDB proceeds at full execution speed during initial execution. URDB uses DMTCP/ptrace to occasionally save a checkpoint image, but this takes about a second, and this runtime over-

head tends to be negligible. The time to find a bug is limited by the time between adjacent checkpoints, but URDB can add additional interim checkpoints by re-executing an execution interval at full speed.

Tralfamadore [LCF+09] represents a somewhat different approach to post-mortem debugging. In that work, an execution trace is unified with the source code itself. Hence, one begins by examining the code at a high level, looking, for example, for frequent paths through the code. One might then observe a less frequent path through the code, and "drill down" eventually to a single execution trace to determine if an unusual execution trace represents a bug.

3. The Architecture of URDB

3.1 Checkpointing with DMTCP

URDB is based on DMTCP/ptrace. DMTCP is free software distributed from `http://dmtcp.sourceforge.net`. It was developed over five years [ACA09, CAM06, RAC06], and as of this writing, it is experiencing approximately 100 downloads per month. Dynamic libraries are saved and restored as part of the user-space memory. A target application can autonomously request its own checkpoint with the *dmtcpaware* library API.

```
dmtcp_checkpoint ./a.out
dmtcp_command --checkpoint
   [ Kill a.out and other processes
spawned by it. ]
dmtcp_restart ckpt_*.dmtcp
```

Figure 1. DMTCP Usage. The command `dmtcp_checkpoint` automatically creates a centralized coordinator process on the local host if the user has not explicitly created one with the command `dmtcp_coordinator`. The commands `dmtcp_checkpoint`, `dmtcp_restart` and `dmtcp_command` can use a centralized coordinator on a specified host via command line flags or environment variables.

DMTCP employs a centralized coordinator, to which each process of a distributed computation connects. The centralized DMTCP coordinator is *stateless*. If the coordinator process dies, then one kills the other processes of the computation and starts a new DMTCP coordinator. Processes are restarted from the *checkpoint images*: `dmtcp_restart ckpt_*.dmtcp`. *Process migration* is accomplished by moving the checkpoint images to new computer nodes.

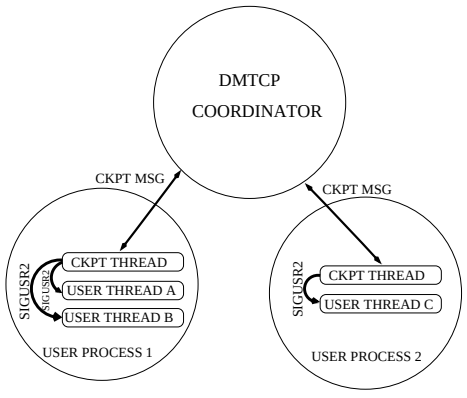

Figure 2. The Architecture of DMTCP

Figure 2 pictorially describes the architecture of DMTCP with two user processes: user process 1 has two user threads A and B and process 2 has one user thread C. Both user processes have one checkpoint thread. The DMTCP coordinator sends checkpoint or restart messages to the two user processes through the two checkpoint threads.

Three basic requirements of distributed, user-space checkpointing are accomplished:

1. restoring user space memory;
2. restoring kernel status; and
3. restoring network data in flight.

The centralized server is needed: to enforce the distributed barriers; and to act as a nameserver where end-user processes register themselves upon restart, in order to discover their peers for reconnecting socket connections.

Among the features automatically accounted for by DMTCP are:

- fork, exec, ssh, mutexes/semaphores, TCP/IP sockets, UNIX domain sockets, pipes, ptys (pseudo-terminals), terminal modes, ownership of controlling terminals, signal handlers, open file descriptors, shared open file descriptors, I/O (including the readline library), shared memory (via mmap), parent-child process relationships, pid and thread id virtualization.

The technical implementation uses `LD_PRELOAD` to preload a DMTCP library, dmtcphijack.so, into each application process. The preloaded library creates an additional checkpoint thread for each application process. The checkpoint thread creates a connection to the centralized coordinator and listens for commands. Signals and a signal handler are used for the checkpoint thread to capture the "attention" of the end-user threads. Wrappers around the fork and exec system calls are used to ensure that dmtcphijack.so is preloaded into child processes. A wrapper around exec captures any calls to "ssh". Upon `ssh exec ...`, it preloads dmtcphijack.so into the new, remote processes. DMTCP implements wrappers for the dynamic library *libc.so*. Further implementation details are in [ACA09].

3.2 The Debugger Monitor

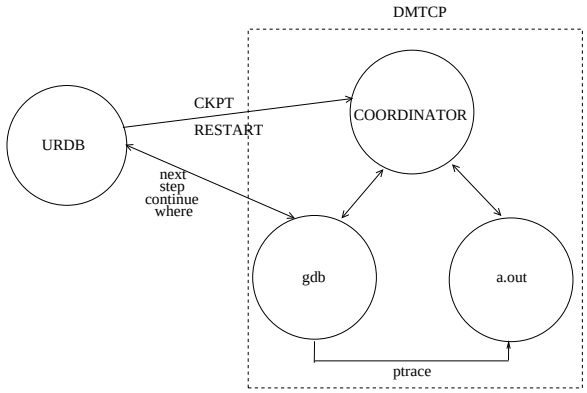

Figure 3. The Architecture of URDB

The URDB package consists of the DMTCP checkpoint-restart package and a python-based debug monitor. Figure 3 describes the architecture of URDB, using gdb as an example. URDB currently supports the following target debuggers: gdb, MATLAB, python (pdg) and perl (perl -d). Additional debuggers can be supported by filling in a python-based template of about 250 lines with the names of debugger commands and regular expressions to capture their output format. URDB communicates with both the coordinator (to

pass on checkpoint and restart commands) and the debugger (to pass on debugger commands).

URDB also uses DMTCP's support for checkpointing ptrace-based applications. Gdb uses ptrace, while MATLAB, python and perl do not.

The debug monitor sits between the end user and the target debugger. For the most part, it passes user commands to the debugger and returns the debugger output (including interrupts ($\hat{C}$) by the user). Certain URDB commands are intercepted by the debug monitor, whereupon the debug monitor converses with both the target debugger and the DMTCP checkpoint-restart system. The target debugger is assumed to support four standard debugger commands: `next` (do not step into any function calls); `step` (step into a function call); `breakpoint` (set a breakpoint); and `continue` (until next breakpoint). Where the debugger also provides `finish` (until end of function), URDB will support the use of that command by the end user. The history of executions of these commands is recorded in a user command history.

The debug monitor commands supported are: `checkpoint`, `restore`, `undo-command`, `reverse-next`, `reverse-step`, `reverse-continue`, `reverse-finish`, and `reverse-watch`. Undo-command undoes the last command to the debugger. Reverse-next (or reverse-step) takes the process backwards in time to the earliest statement such that a "next" debugger command (or "step" debugger command) would return the process to the current state. (If the process is at the first statement of a function, then reverse-next returns the process back to the statement that called the current function.) Reverse-finish returns the process to the statement of the function that called the current function. For clarity of exposition, we speak of statements, but in fact, a *statement event* is intended, the last time prior to the present when the indicated statement was executed.

The last command, `reverse-watch`, is discussed in detail in Section 4.

This is intended as a more general variation of data watchpoints, since the expression may depend on many addresses in RAM. While some standard debuggers also provide expression watchpoints, the URDB approach is more efficient, since standard debuggers must evaluate the expression before each of the statements. See Section 4 for a fuller description.

The debug monitor spontaneously generates additional checkpoint images (not necessarily requested by the user) when the time since the last checkpoint grows beyond a specified threshold. To reduce the disk space consumption by the checkpoint files, we employ an exponential aging algorithm, as described in Section 4.

Timeout for "continue". Since the command `continue` followed by `reverse-step` is problematic if the "continue" command had executed for a long time, URDB places a timeout on "continue". If "continue" executes beyond the timeout interval, then URDB creates an additional checkpoint. URDB will proceed with the "continue" command, after the checkpoint.

The timeout interval is chosen so as to maintain a reasonable user response time for URDB. The "continue" command executes natively in many debuggers, while the "step" command is interpreted. Hence the specified timeout interval guarantees that if a single "continue" command is expanded into a series of "step" commands, then the resulting interpreted execution (step-by-step) will complete in reasonable time. This turns out to be important to guarantee reasonable response times for the algorithms below.

Algorithmic notation. The algorithm can be expressed more clearly using a modified rewrite rule notation.

A *user command history* is available at each moment when the target debugger is stopped. The command history is maintained for three target debugger commands: "step", "next", and "continue" (and optionally "finish"). The commands are notated by the four characters, `s` (step), `n` (next), `c` (continue), and `f` (finish), respectively.

A regular expression-like notation is used for command histories:

- "`*`", 0 or more occurrences of a target debugger command.
- "?", exactly one target debugger command.
- "`.`", separates tokens syntactically and has no further significance.

Hence, a command history `*.c.n` denotes an arbitrary sequence of commands followed by `continue` and `next`.

A rewrite rule, `*.n` $\rightarrow$`*`, denotes to match a command history for which the last command was `next`. If a match is found, then truncate the final "next" command from the command history, and use the checkpoint-restart facility of DMTCP to re-execute the debugging session from the last checkpoint until just before the "next" command was issued. Similarly, a rule `*` $\rightarrow$`*.c` says to simply execute the "continue" command in the target debugger. Thus, the URDB command `undo-command` would be algorithmically described by `*.?` $\rightarrow$`*`.

Process state. At any given time, the target application being debugged is in a unique *process state*. Here are the states a process can be in:

- `ORIG_STATE`: the state of the process at the time that a URDB command is issued.
- `DEEPER`: if the stack depth (number of call frames on the stack) at the current time is greater than the original stack depth (depth at the time of `ORIG_STATE`).
- `SHALLOWER`: if the stack depth is less than original stack depth.
- `SAME`: if the stack depths are equal. The `SAME` state come in two flavors:
  - `SAME/ORIG_STATE`: similar to `ORIG_STATE`.
  - `SAME/NOT_ORIG_STATE`: a prior time in the process history, since the algorithm never travels forward in time beyond `ORIG_STATE`.

As rewrite rules are executed, the process will travel backward and forward in time, and the command history will be adjusted accordingly. The URDB algorithm never travels beyond `ORIG_STATE` (see Section 3.3 for a description of how `ORIG_STATE` is detected).

As the rewrite rules execute, the process can be any of the following four states: `DEEPER`, `SHALLOWER`, `SAME/ORIG_STATE` and `SAME/NOT_ORIG_STATE`. This is important, since the current state will determine which set of rewrite rules should be applied.

The algorithms. Each algorithm presented in this section begins execution in state `SAME/ORIG_STATE`. Rewrite rules from the current state are executed until a "Return" statement is encountered. If two rewrite rules within a given state both match the current user command history, then only the first of the two rewrite rules is applied.

The notation `STATE(*` $\rightarrow$ `*.n)` denotes the state that would result from executing the indicated rewrite rule. If a rewrite rule contains an *action* in square brackets, then the action is executed after the rewrite rule is executed.

A function `LEVEL(*.n)` indicates the stack depth of the user command history from the corresponding rewrite rule.

The `FAIL` statement denotes that the reverse command could not be executed.

Both algorithms for `REVERSE_NEXT` and `REVERSE_STEP` use `BACK_UP_TO_SAME`. Function `BACK_UP_TO_SAME` brings the current

number of call frames on the stack to the same number as in ORIG_STATE.

```
BACK_UP_TO_SAME () :
  case DEEPER:
    *.? → *
  case SHALLOWER:
    * → *.c
  case SAME:
    Return
```

Both algorithms for REVERSE_NEXT and REVERSE_STEP adjusts the number of call frames on the stack to the same number of call frames as ORIG_STATE and then according to the last command in the command history, the algorithms decide how many and which commands of the commands history to execute.

```
REVERSE_NEXT ( ORIG_STATE ) :
  case DEEPER:
    If STATE(* → *.n) = ORIG_STATE, then:
      BACK_UP_TO_SAME()
    * → *.n
  case SHALLOWER:
    * → *.s
  case SAME:
    case ORIG_STATE:
      *.n → * [If LEVEL(*.n) = LEVEL(*), then Return]
      *.s → * [If LEVEL(*.s) >= LEVEL(*), then Return]
      *.? → *
    case NOT_ORIG_STATE:
      * → *.n


REVERSE_STEP ( ORIG_STATE ) :
  case DEEPER:
    If STATE(* → *.n) = ORIG_STATE, then:
      BACK_UP_TO_SAME()
    * → *.n
  case SHALLOWER:
    * → *.s
  case SAME:
    case ORIG_STATE
      *.s → * ; then Return
      *.? → *
    case NOT_ORIG_STATE
      If STATE(* → *.s) = ORIG_STATE, then:
        * → *.s
      Else if STATE(* → *.n) = ORIG_STATE, then:
        * → *.s
      Else:
        * → *.n
```

The algorithm for REVERSE_CONTINUE is presented next. From the beginning of the computation until the time the user requests a reverse-continue, N checkpoints were taken. Checkpoint i is represented by C_i. The checkpoints sequence is the following: C_1, C_2, ..., C_N. C_1 is the first checkpoint taken, whereas C_N is the last one.

The algorithm for REVERSE_CONTINUE starts with i initialized to N (the last checkpoint).

```
REVERSE_CONTINUE ( ) :
  BEGIN: Restore last checkpoint C_i
  Repeat continue until ORIG_STATE
  Let B be the number of breakpoints found
  If more than one breakpoint found
      Restore last checkpoint C_i
      Re-execute until breakpoint B − 1
  If no previous breakpoints found
    Set ORIG_STATE to state (time) as of C_i
    Set i to i − 1
    If i = 0, FAIL
    Ifelse goto BEGIN
```

The algorithm for REVERSE_FINISH is defined as going backwards in time to the site at which the current function was called. The following scenario is presented: a function A is calling function B. The user decides to issue a reverse_finish while inside function B. reverse_finish will take the program to the site where function B was called.

If the target debugger implements the $finish$ command, then URDB supports the REVERSE_FINISH command.

```
REVERSE_FINISH ( ORIG_STATE ):
  Repeat
    REVERSE_NEXT()
  until reaching SHALLOWER level.
```

The undo-command k command is implemented in the following way. When a user requests undo-command k for some integer k, URDB restores the last checkpoint and then re-executes the first $n - k$ of the user commands since the last checkpoint.

All algorithms presented in this paper allow for multiple checkpoints. For example, if a user is currently at checkpoint C_5 it is possible to reach C_1 by issuing a sequence of URDB reverse commands.

reverse-continue The reverse-continue command is implemented in one of two ways. If the time since the last checkpoint is small enough, then repeated continue commands are executed from the last checkpoint. (A breakpoint is set at the current position so that the checkpoint does not go beyond the current time.) A count of the number n of continue commands is maintained. Upon discovering the last breakpoint, prior to the current time, the process is re-executed with $n - 1$ continue commands.

If no breakpoints were hit between the last checkpoint and the current time, then the checkpoint prior to that one is restored, with the "current time" being set to the last checkpoint. The reverse-continue algorithm is then repeated.

Finally, it can happen that the time from the last checkpoint to the current time is above the specified threshold. In that case, a binary search strategy is employed (see Section 4), with the condition of the binary search being to test whether a breakpoint was seen. If a breakpoint is found in both the first and second half of a binary search, search recursively explores the second half, in order to determine the last breakpoint encountered. When the time interval being explored falls below a specified threshold, the reverse-continue algorithm falls back to the previous strategy of repeated continue commands.

3.3 Implementation of the Debug Monitor

The debug monitor uses the concept of breakpoint events. A *breakpoint event* is a process state (time) at which a breakpoint was hit. If one hits the same breakpoint multiple times, one must distinguish the different breakpoint events. In gdb and some other debuggers, one can determine a *unique* breakpoint event as the debugger travels backward and forward in time. One does this by noting the number of times that a breakpoint was hit by the debugger. For example, gdb reports that number through the gdb command info breakpoints.

For debuggers that do not associate a number with each breakpoint, the filename and line number of that breakpoint serve the

same purpose. Some debuggers do not record the number of times a particular breakpoint was hit. In those cases, URDB increments a variable of the target debugger (or even a global variable in the target application) with the total number of breakpoints hit so far. As the process moves backward and forward in time, this number is automatically updated. In this case, a breakpoint event is simply the number of breakpoint hits encountered since the beginning of the process.

A key to the URDB algorithm is being able to recognize the `ORIG_STATE` of the preceding section. The implementation defines a process state to be a triple: (filename, line number, last breakpoint event seen). The current process state is considered to be in the `ORIG_STATE` when it has the same value of the triple. There may be more than one process state having the same triple. The key to the correctness of the algorithm is that when the current process state has the same triple as the original process state, then the two process states must have the same command history through the last "continue" statement. Hence, any potential differences in the command history concern only sequences of "step" and "next" instructions, which are directly handled by the algorithm.

As an optimization, coalescing of debugger commands such as "next" and "step" is employed. For example, a debugger command "next; next; next" can be replaced by "next 3" for greater efficiency. The issue of debugger breakpoints adds a subtle point to the implementation. All breakpoints must be temporarily disabled or deleted during a repeated "next". Yet breakpoints and disable/delete commands in the user history must continue to be faithfully re-executed.

4. Temporal Search: Reverse Expression Watchpoints via Binary Search

Reverse expression watchpoints are implemented as a showcase for the method of temporal search. Additional temporal search methods can be added in the future.

Consider the following scenario. During a debugging session, a user has isolated a bug. They print a certain expression within the debugger, and observe that their expression has an incorrect value. They know that at some point in the past, the value of the expression had been correct. So, there must have been at least one point of time (and possibly many) where executing a statement caused the value of the expression to change from a correct value to an incorrect value.

Each example in which the statement causes the expression to change from correct to incorrect is an example of a fault. But those faults did not expose the bug as an error, until much later in the execution. So, the user needs to move backwards in time from the error to a previous fault.

The URDB command *reverse_watch* automates this search. It creates a *reverse expression watchpoint* for the expression in question. It then brings the user directly to a statement (one that is not a function call) and event at which the expression is correct, but execution of the statement will cause the expression to become incorrect. The user may then use standard debugger commands to determine the cause of the fault. (It is not important which of the several possible faults is diagnosed. Any fault at all is part of a bug.)

The mechanism that `reverse_watch` uses to find the statement corresponding to the fault is essentially a binary search along the process lifetime. Some implementation details are provided in Section 4.4.

The term *reverse* is used to emphasize that the expression to search on is declared only after the program has finished executing the region of interest. The term *expression watchpoint* is used to emphasize that this is a generalization of the well-known data watchpoints supported by many debuggers. Debuggers tend to support only data watchpoints and not expression watchpoints, for efficiency reasons. For data watchpoints, virtual memory hardware is used to efficiently stop the program when the value at a data address of interest changes.

Since a general expression does not correspond to any single address in memory, this technique is either not extended to expression watchpoints, or else expression watchpoints are supported through the extraordinarily slow mechanism of executing the desired expression at *each and every* program statement. URDB supports the even more powerful reverse expression watchpoints, while remaining consistent with its objective that code always execute at full native speed.

4.1 Example: phase change computations

The motivation for *reverse expression watchpoint* can be given through phase change computations. Consider the case of random graphs, constructed by the addition of new edges at random. The user wants to know when a cycle in the graph first appears, when the graph becomes "mostly connected", when cliques above a certain minimum threshold first appear, etc. Often such changes appear throughout the graph at approximately the same time. In such cases, the time when they appear is known as a *phase change*.

Such phase change problems appear throughout the sciences and social sciences. Some typical examples are programs that analyze graphs from social networking, from the analysis of the structure of the Internet, from physics percolation problems, from mathematical studies of random graphs, and from a wide variety of other problems. In order to determine the average number of edges at which the phase change appears, and the standard deviation, the user runs many simulations. In each simulation, edges are added to a graph, and the user must write sophisticated code to incrementally update information about the graph to determine when the property first appears.

Instead of writing sophisticated code to incrementally update information on the graph, the user can instead write a naive routine, `propertySatisfied()`, which tests if the phase change has occurred yet. The routine `propertySatisfied` returns `true` or `false`. A call to this routine becomes the expression in the reverse expression watchpoint. Near the end of the simulation, `propertySatisfied` returns `true`. Near the beginning of the simulation, it returns `false`. If N edges are added, then $\log_2 N$ binary search probes suffice. So, `reverse_watch` calls `propertySatisfied` $\log_2 N$ only times. Even for huge graphs, this will typically be less than 40 times.

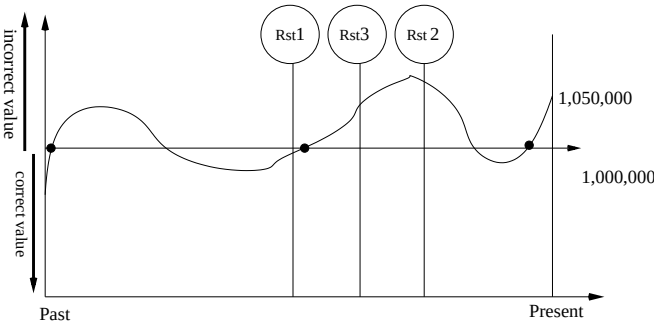

Figure 4. Reverse-expression Watchpoint example for a bounded linked list

4.2 Example: linked list

Another example is a bounded linked list on which we perform insertions and deletions. By inserting and deleting, the length of

the linked list will change in a non-monotonic fashion. Also, consider that we are performing about 1,000,000 operations (insert or delete) per minute and the linked list cannot exceed 1,000,000 elements. Figure 4 illustrates this example. We notice that in Figure 4 the bug is exhibited 3 times (marked by the filled black circles). However, the bug is observed only the third time. The user wants to know when the length of the bounded linked list grew greater than 1,000,000 elements. In this example, we notice something interesting. The *reverse expression watchpoint* algorithm does not return the place where the bug occurred for the first time. Since we are fixing bugs, any occurrence is just as good.

4.3 Cost of Executing a Reverse Expression Watchpoints

In the two previous examples, the expression under consideration is an especially expensive one. Assume that the program executes N statements. While $\log_2 N$ calls to the expression of interest (e.g., `propertySatisfied`) may be acceptable, some number of calls proportional to N is not acceptable. In fact, even when the expression is just the sum of two variables, evaluation of that expression is surprisingly expensive under gdb. This is because gdb must call ptrace twice, invoking the operating system each time. The operating system then *peeks* into the memory of the inferior (target) process at the requested address, and returns the value to the superior (gdb) process.

It is clear that the time for execution of a reverse expression watchpoint over an interval of N statements consists of:

1. $\log_2 N$ restarts from a checkpoint;
2. $\log_2 N$ iterations re-executing the code from the checkpoint until the current bisection point specified by the binary search; and
3. $\log_2 N$ evaluations of the expression at the current bisection point.

It is important to the end user that the execution time for a reverse expression watchpoint always be within reason for an interactive debugger. The user wishes to try an expression under the reverse expression watchpoint, and then try another one if the first expression does not uncover the cause of the bug.

For the sake of specificity, assume that $\log_2 N = 40$. On a machine executing a million statements per second, this corresponds to 11.5 days worth of execution. Since a restart executes in less than a second, and hopefully the user evaluation of the expression executes in less than a second, components 1 and 3 of the time above each consume at most 40 seconds — even for our hypothetical case of a run over 11.5 days.

This leaves the second component, the time to re-execute the code. If this were to take 40×11.5 days, that would be unacceptable. Luckily, it is possible to create temporary checkpoint images at the end of each iteration. If we create an extra temporary checkpoint image at the end of each iteration of the binary search, we have 40 checkpoint images. The cost of 40 additional checkpoints contributes an additional 40 seconds. (In fact, once the interval of execution for the binary search is less than a second, we stop creating temporary checkpoint images. So, there are only 20 temporary checkpoint images in this scenario.)

By using these additional checkpoint images, we can guarantee that for each iteration of the binary search, the time to re-execute the code will be half of the time during the previous execution. The sum $1/2 + 1/4 + 1/8 + \cdots$ is equal to just one. So, the total time of the second component (re-executing the code) is no more than the original 11.5 days to execute the code. This keeps URDB true to its goal of always executing code at full native speed.

4.4 Implementation of Reverse Expression Watchpoints

URDB has two modes of operation for binary search: **command history-based binary search** and **time-based binary search**.

For **time-based binary search**, on each iteration of the binary search executes for half of the total time duration between the start point and the process state at which the reverse expression watchpoint command was given. It then creates a temporary checkpoint image, as described in the previous section. Once the time interval of the binary search is reduced below a specified threshold, URDB switches to a command history-based binary search.

For **command history-based binary** search, we continue the binary search, but we now use the history of debugging commands that were originally invoked by the user. Now, instead of cutting the time in half on each iteration, we cut the number of debugging commands to be executed in half.

We also take advantage of a natural hierarchy of debugging commands. In URDB, this hierarchy consists of commands to:

1. issue a "continue" command until reaching the next checkpoint image (in fact, this is optimized to simply restart from the next checkpoint image);
2. issue a "continue" command until the next breakpoint;
3. issue a "next" command that *steps over* any function calls; and
4. issue a "step" command that *steps into* any function calls.

Note that a "continue" command may terminate either upon reaching a checkpoint, or upon reaching a breakpoint. It may terminate upon reaching a checkpoint because the user had originally issued a "continue" command that took many seconds to complete. In that case, URDB inserts an extra checkpoint in the middle, and then restarts from that extra checkpoint by issuing an additional "continue" command. An aging process is used to ensure that there are not too many successive checkpoints followed by "continue".

Finally, when the binary search reduces to search over a command history of length one, then URDB expands that single debugging command into repeated debugging commands at the next lower level of the hierarchy. A single "continue" command expands into repeated "next". A single "next" command expands into repeated "step".

There is a subtle issue for the case of a "continue" command terminated by a checkpoint. If the command history reduced to a single "continue" command terminated by a checkpoint, then that command is expanded into repeated "next". How does one know how many repeated "next" commands are needed to reach the next checkpoint? The answer is that one doesn't know.

Then one resorts to a more expensive heuristic — use a fine-grained timer, and continue to bisect the interval into successively smaller fractions of the original one-second interval. Upon reach a millisecond, expand the remaining interval by repeated "next" and evaluate the expression of interest after *every* "next" command to find when it changes from `true` to `false`.

A similar observation applies to expanding "next" into repeated "step" commands. The heuristic may be needed in this case if the program under examination is highly recursive.

5. DMTCP/ptrace

DMTCP/ptrace was an essential requirement in order to demonstrate the principles of temporal search in a C/C++ environment using gdb. The extension of DMTCP to DMTCP/ptrace is new with this work.

Ptrace is a Linux system call. It allows a superior process (gdb in our case) to trace an inferior process (target application) at the binary level. The ptrace system call includes several sub-commands: peek at memory of target, poke into memory of target, step through

one machine instruction, and continue until target sees next ptrace-event. A ptrace-event for the inferior process consists of the inferior process receiving a signal. The signal can come from a different process, but it can also be a synchronous event if generated by a trap machine instruction (software interrupt).

Conceptually, gdb can be viewed as a front-end for the functionality provided by ptrace. Gdb commands such as stepi, continue, and the ability to stop an inferior process by sending an interrupt (control-C), all reflect the functionality provided by ptrace. Additionally, gdb supports breakpoints and source-level debugging (statement at a time) by inserting trap machine instructions into the binary code. When the trap instruction is executed, it generates a software interrupt, which in turn is converted into a signal, and so is recognized as a ptrace-event. This halts the inferior process, which is then available for further commands from the superior process (gdb).

DMTCP/ptrace was enhanced with a wrapper around the "ptrace" system call. The wrapper is required to record the superior — inferior pairs, information which becomes important at resume (after a checkpoint) or restart time.

While the checkpoint file is written to disk, the inferior process is no longer traced by the superior process. The reasoning behind this is as follows: while the inferior process is being traced by the superior process, it is waiting for commands from its superior. During checkpoint the control is within DMTCP. Once checkpointing has begun, the superior process cannot issue ptrace commands to the inferior process. This means that in the absence of ptrace commands from the superior, the inferior process cannot write the checkpoint file to disk.

At resume time, once the checkpoint is complete, the superior process starts tracing the inferior process. At restart time, the processes need to know if they are tracing any inferior processes and which are the inferior processes they are tracing. Both during resume and restart time, the information recorded in the "ptrace" wrapper is used.

Some of the difficulties we faced in checkpoint/restarting ptrace are presented next:

1. *eflags register.* The ptrace system call is in turn based on the eflags hardware register of the x86 architecture (both for 32-bit and 64-bit CPUs). Once the superior process starts tracing the inferior process, the eflags trace bit is set for the inferior process. One of the issues we had to face was the fact that at restart time, the eflags trace bit was no longer set.
2. *Tracing the checkpoint thread of the inferior process.* gdb, the superior process, is tracing each thread of the inferior processes (via PTRACE_O_TRACEVFORK request). Since we are under the control of DMTCP, a checkpoint thread is created in both the superior and inferior process. The checkpoint thread is responsible for processing checkpoint and restart message from the coordinator. The inferior checkpoint thread could not process the messages from the coordinator, if traced by gdb.
3. *Inside DMTCP code at restart time.* Upon restart by DMTCP, a process will begin life inside DMTCP's own signal handler. This is not user code. The superior process needs to set the eflags trace bit and single-step the inferior process out of the signal handler into user code.
4. *Tid virtualization.* Still another issue was the need to implement tid (thread id) virtualization in DMTCP. gdb sends a ptrace command to a specific inferior (specific tid). Upon restart, each thread is given a new thread id by the operating system. So, a thin virtualization of the relevant system calls was needed in order to translate between the thread's original tid, and the thread's current tid after restart. The original tid becomes the virtual tid, while the current tid is the actual tid known to the operating system kernel.

6. Experimental Results

Unless otherwise indicated, all experiments are done under Ubuntu 9.04 with a Linux kernel 2.6.31. The computer used has an ntel Core 2 Duo CPUs running at 3.0 GHz. The experiments used gcc 4.3.3, gdb-6.8, python 2.6.4, perl 5.10.0 and MATLAB 7.4.0. MATLAB was used on a four-core Intel Xeon running at 1.86 GHz. The version of DMTCP used was revision 512 (unstable branch with support for ptrace) of the DMTCP subversion repository. URDB revision 246 of the URDB subversion repository was used. DMTCP was configured to not use compression. This produces larger checkpoint images, but checkpoint and restart are faster.

URDB is entirely implemented in python. The python monitor component contains 2,300 lines of code. URDB also requires a debugger-specific python file for each of its four target debuggers. The python-based personality template expands to a debugger-specific file containing: 220 lines of code for MATLAB; 270 lines of code for python (pdb module); 260 lines of code for perl (perl -d); and 340 lines of code in the case of gdb. DMTCP/ptrace consisted of an additional 1,850 lines of code for ptrace.

The experiments are grouped into four parts:

1. Bisimulation Bugs
2. Debugging graphics applications
3. Experiments across reversible debuggers
4. Debugging BSD UNIX games

Bisimulation Bugs The modification to open source software may introduce some bugs. To pin down this kind of bug, "outsiders" generally compare the new version with the original version following the black-box method. We take the Geant4 multithreading work as an example [CD08] to illustrate how the temporally reversible debugging boosts the debugging by comparison.

Geant4 [Gea] is a toolkit that provides a Monte Carlo simulation of particle-matter interaction for high energy physics. It was designed using the Booch (Unified) methodology, and documented using Booch/UML notation [Cf01]. It is a 750,000 line toolkit first designed in the mid-1990s and originally intended only for sequential computation. Intel's promise of an 80-core CPU meant that Geant4 users would have to struggle in the future with 80 processes on one CPU chip, each one having a gigabyte memory footprint. Thread parallelism would be desirable.

Although the event loop style of Geant4 lends itself to a semi-automatic parallelization methodology, there still remain some non-trivial issues of parallelism. In order to eliminate possible bugs injected by the semi-automatic transformation, we compare the original Geant4 program with the multithreaded version to verify whether they have the identical behavior in the simulation phase. This is exactly the proof by the bisimulation where the original Geant4 program is regarded as the "specification" and the multi-threaded version is regarded as the "implementation".

Without the temporally reversible debugging tool, we have to start two versions of applications, break where we suspect the problem is and step into the two programs for the comparison. Each round of debugging session requires more than 15 minutes for the initialization to complete. Then we can do the comparison. Moreover, many rounds of debugging are required in order to guess where the bug resides, until we capture the bug.

With the temporally reversible debugging tool, we checkpoint the two programs after the initialization and later, at a point in time where the two programs have an identical behavior. In each debugging round, we start from the most recent checkpoint. This allows us to move forward in big steps. After we restart from a

checkpoint, the user issues a debugger command. Once the command completes, the user checks the state of the two programs. If both programs are in the same state, continue. If they are in different states, restart from a checkpoint and issue a different debugger command. This increases the convergence speed tremendously in the effort to capture the bug from originally one week to currently several hours.

Debugging graphics applications. Temporal debugging is a very helpful tool in debugging X-windows applications, using a GUI interface. X-windows applications require user input. Different user inputs corresponding to keyboard or mouse events sequences lead to different execution paths. Given this scenario, URDB allows the programmer to take checkpoints of the program state to speed up the debugging process. Checkpoints are especially efficient when the bug is noticed after a long interaction time.

We have successfully debugged xcalc 1.0.1 (an X-windows application using a GUI interface) using URDB. The xcalc 1.0.1 bug is mentioned at [xca], when emulating the HP-10C. The e^x button exhibits the following bug: when typing a large number, e.g. 800 and then clicking the e^x button, the value returned is "error" instead of "inf". If the button e^x is clicked one more time, the result is "inf".

In order to debug this issue, we used URDB. We set up a breakpoint and took a checkpoint after the button e^x is clicked. Then we stepped into the function corresponding to the e^x button. e^{800} is computed as "inf" by xcalc 1.0.1. This result is further used as the input for the second time we click e^x with the final result as "inf". The reason "error" is displayed after the first time we click the e^x button is because "errno" is set up to a non-zero value. The second time we click the e^x button, "errno" is set to zero.

xcalc 1.0.1 uses the *exp* C function, that does not set the variable "errno" to ERANGE if the input is already infinite.

Using URDB we could find the source of the bug in a few minutes.

Experiments across reversible debuggers. URDB can easily be ported to different debuggers. To illustrate this, we added the following four personalities to URDB: gdb 6.8, MATLAB, python(pdb) and perl (perl -d).

The four different personalities were tested on the example of an acyclic random graph, to which random edges are added. After each edge is added, the graph must remain acyclic. However, testing if a graph is acyclic or not after each edge is added is a very expensive operation, especially for large graphs. A much better option is using *reverse expression watchpoint*.

The graphs we tested on are presented in Table 1. Since C/C++ is a compiled language, the time to generate the edges is extremely fast, as compared to interpreted languages, like MATLAB, perl, python.

The number of edges per second was computed by dividing the number of edges added by the time taken to insert the edges (see Table 1, column 3). The time to run the cycle detection algorithm is given in Table 2, column 2. Given the big number of edges inserted per second and the long time required to detect the existence of cycles, current solutions (like checking for cycles after each edge insertion) fail to be feasible (see Table 2, column 4).

Language	nodes	edges	edges per second
C/C++	1,000	200,000	3,703,300
MATLAB	5,000	6,000,000	430,000,000,000
perl	5,000	6,000,000	150,000
python	5,000	6,000,000	48,000

Table 1. Number of nodes and edges of the graphs used, as well as the number of edges added per second.

Table 2 shows the time when the user tests for the existence of a cycle for the first time, the time required by the cycle detection algorithm to complete, the time needed to determine the edge that created a cycle, using *reverse expression watchpoint* and the time to execute the current algorithm. The *reverse expression watchpoint* is obviously much better.

Debugger	until cycles	cycle detection	reverse watchpoint	current algorithms
gdb-6.8	0.054	0.1	17.8	20,000
MATLAB	0.000014	1	54.4	6,000,000
perl -d	40	70	110.6	4,200,000,000
pdb	123	15	113.4	90,000,000

Table 2. Time until the user checks for cycles, the time to run the cycle detection algorithm and the time to determine the edge that created a cycle via reverse-watchpoint (seconds).

Table 3 shows the number of checkpoints taken until the user checked for cycles, as well as the average size of a checkpoint file.

Debugger	number of checkpoints	average checkpoint size
gdb-6.8	30	1,600
MATLAB	6	41
perl -d	6	558
pdb	6	120

Table 3. Number of checkpoints taken during the experiment and the average checkpoint size(in MB)

Table 4 reports on the underlying average times to checkpoint and to restart for the above experiment for DMTCP alone (without using URDB).

Debugger	checkpoint	restart
gdb-6.8	0.81	0.42
MATLAB	7.19	2.92
perl -d	0.43	0.20
pdb	0.53	0.19

Table 4. Times for checkpoint and restart (seconds)

In the case of gdb, there are two checkpoint images: gdb (9.5 MB); and the target application a.out (1.6 GB). MATLAB employs two checkpoint images: MATLAB (30.3 MB); and matlab_helper (2 MB). Python and perl consist of a single checkpoint image each of sizes 3.66 and 3.42 MB, respectively. In the case of gdb and MATLAB, Table 4 reports the combined average sizes of the two checkpoint images.

BSD UNIX curses-based Games. URDB was also tested on two classic 1980-era BSD UNIX curses-based games, robots and rogue, were chosen to demonstrate this category. (See the Wikipedia articles "Rogue (computer game)" and "Robots (computer game)".) A modified URDB was employed with control-E/control-T for checkpoint/undo, respectively, since interactive games do not read commands through standard input.

The version of "robots" used was version 2.16 of the bsd-games Ubuntu package, while "rogue" was version 2.17 of the bsdgames-nonfree Ubuntu package. The times for checkpoint and undo for robots were 3.67 and 0.04 seconds, respectively. Similarly, the times for checkpoint and undo for rogue were 3.21 and 0.05, respectively. The checkpoint images were of size 12.5 and 12.8 MB for robots and rogue, respectively.

The experimental results validate the temporal debugging approach. The single largest threat to the validation is the case of

debugging compiled code for short intervals of time. This represents the limiting case where the code runs fastest (since it is compiled and since the time interval is short). At the same time, the short time interval continues to require the overhead of a modest minimum number of checkpoints and restarts via DMTCP/ptrace. The time for checkpoint and restart is exacerbated further for programs with a large memory footprint (for example, one gigabyte), where the time to store a checkpoint image on disk is significant. In contrast, for long-running compiled languages, and for interpreted languages over all time intervals, the temporal debugging approach shows clear advantages. As described in the next section, an incremental checkpointing ability is planned for DMTCP/ptrace as part of the future work, in order to reduce or eliminate this bottleneck.

7. Conclusion and Future Work

We have seen that URDB succeeds in finding faults corresponding to errors in long-running programs. The methodology has been demonstrated across four widely different debuggers. Supporting gdb and C/C++ was the most challenging, since it required the implementation of DMTCP/ptrace to support checkpointing of gdb debugging sessions.

As part of its design goal, URDB runs essentially at full native speed, except when checkpointing or restarting a debugging session. For comparison, as seen in Section 2.1, gdb runs about 100,000 times slower than native speed during intervals when instruction logging is turned on. The gdb instruction logging approach also suffers from space problems, since it consumes about 100 bytes per C statement in our experiment.

The demonstration of *bisimulation* for a real-world program, Geant4, represents a useful extension of traditional reversible debugger. It allows one to move forward and backwards through two versions of the same program in tandem. This is important in situations where complex software goes through multiple revisions, and regressions or other bugs can appear.

Incremental checkpointing is planned for a future version of URDB in order both to reduce the size of each checkpoint and to improve the times for checkpointing. This will have the largest impact on the use of reverse expression watchpoints for compiled languages over short time intervals. In this case, the cost of repeated checkpoints and restarts can dominate over traditional methods. For interpreted languages such as MATLAB, Python and Perl, this time usually does not dominate, due to the greater time of interpreted execution.

Finally, expression watchpoints are an example of *program-based introspection*: when in my process lifetime did this expression change its value? The example of finding when a cycle appears in a random graph represents a limited example of program-based introspection. Conversely, one can see a limited form of *speculative program execution* in the reverse capability for the interactive games *rogue* and *robots*.

These rudimentary examples open the way in the future for more sophisticated applications of: *time-traveling program-based introspection*; and *speculative program execution*.

8. Acknowledgement

We wish to thank Priyadarshan Vyas for his contributions to an earlier version of the monitor able to support the *undo* command described in this paper. We also thank Shobhit Agarwal and Amruta Chougule for their earlier identification of several issues and suggested solutions for checkpointing gdb with DMTCP. We also wish to thank Peter Desnoyers and Jason Ansel for their helpful discussions and insights.

References

[ACA09] Jason Ansel, Gene Cooperman, and Kapil Arya. DMTCP: Scalable user-level transparent checkpointing for cluster computations, 2009. version also available at `http://arxiv.org/abs/cs.DC/0701037`.

[CAM06] Gene Cooperman, Jason Ansel, and Xiaoqin Ma. Transparent adaptive library-based checkpointing for master-worker style parallelism. In *Proceedings of the 6^{th} IEEE International Symposium on Cluster Computing and the Grid (CCGrid06)*, pages 283–291, Singapore, 2006. IEEE Press.

[CD08] Gene Cooperman and Xin Dong. 116 — Progress report toward a thread-parallel Geant4 (invited talk). 13^{th} Geant4 Collaboration Workshop, Oct. 6–11 2008. `http://kds.kek.jp/confAuthorIndex.py?view=full&letter=c&confId=2027`.

[Cf01] Gabriele Cosmo and (for the Geant4 Collaboration). Software process in geant4. In *Proc. of Computing in High Energy Physices (CHEP-01)*, 2001. `http://geant4.cern.ch/reports/papers/Chep2001/SPI/CHEP01_SPI.ps` or `http://www.ihep.ac.cn/~chep01/paper/8-008.pdf`, Paper 8-008.

[FB89] Stuart I. Feldman and Channing B. Brown. IGOR: a system for program debugging via reversible execution. *SIGPLAN Notices*, 24(1):112–123, 1989.

[GDB] GDB team. GDB and reverse debugging. `http://www.gnu.org/software/gdb/news/reversible.html`.

[Gea] Geant4 Web page. `http://wwwinfo.cern.ch/asd/geant4/geant4.html`, 1999–.

[Gri70] Ralph Grishman. The debugging system AIDS. In *AFIPS '70 (Spring): Proceedings of the May 5–7, 1970, Spring Joint Computer Conference*, pages 59–64, New York, NY, USA, 1970. ACM.

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen (VMware Corporation). Debugging operating systems with time-traveling virtual machines. In *Proc. of 2005 USENIX Annual Technical Conference, General Track*, pages 1–15, 2005.

[LCF+09] Geoffrey Lefebvre, Brendan Cully, Michael J. Feeley, Norman C. Hutc hinson, and Andrew Warfield. Tralfamadore: unifying source code and execution experience. In *EuroSys '09: Proceedings of the 4th ACM European conference on Com puter systems*, pages 199–204, New York, NY, USA, 2009. ACM.

[LDC08] E. Christopher Lewis, Prashant Dhamdhere, and Eric Xiaojian Chen. Virtual machine-based replay debugging, 30 October 2008. Google Tech Talks: `http://www.youtube.com/watch?v=RvMlihjqlhY`; further information at `http://www.replaydebugging.com`.

[LDG+08] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system: release 3.11; documentation and user's manual, 2008. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[PT09] Guillaume Pothier and Éric Tanter. Back to the future: Omniscient debugging. *Software*, 26(6):78–85, Nov./Dec. 2009.

[PTP07] 5. G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proc. 22nd ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA 07)*, pages 535–552. ACM Press, 2007.

[RAC06] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent user-level checkpointing for the Native POSIX Thread Library for Linux. In *Proc. of PDPTA-06*, pages 492–498, 2006.

[SKAZ04] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *In Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 29–44, 2004.

[TA90] Andrew P. Tolmach and Andrew W. Appel. Debugging Standard ML without reverse engineering. In *LFP '90: Proceedings of*

the 1990 ACM conference on LISP and functional programming, pages 1–12, New York, NY, USA, 1990. ACM.

[TA95] Andrew P. Tolmach and Andrew W. Appel. A debugger for Standard ML. *J. Funct. Program.*, 5(2):155–200, 1995.

[URD09] URDB team. URDB: Universal Reversible Debugger, 2009. software available at `http://urdb.sourceforge.net`.

[xca] xcalc man page. `http://fts.ifac.cnr.it/cgi-bin/dwww?type=runman&location=XCALC/1`.

[Zel73] M. V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566, 1973.